Proving Correctness of Transformation Functions in Real-Time Groupware

Abdessamad Imine, Pascal Molli, Gérald Oster, Michaël Rusinowitch

Proving Correctness of Transformation Functions in Real-Time Groupware

Abdessamad Imine, Pascal Molli, Gérald Oster and Michaël Rusinowitch
ECOO and CASSIS Teams - LORIA France
{*imine,molli,oster,rusi*}*@loria.fr*

Abstract. Operational Transformation is an approach which allows to build real-time groupware tools. This approach requires correct transformation functions. Proving the correction of these transformation functions is very complex and error prone. In this paper, we show how a theorem prover can address this serious bottleneck. To validate our approach, we verified the correctness of state-of-art transformation functions defined on Strings with surprising results. Counter-examples provided by the theorem prover helped us to define new correct transformation functions for Strings.

Introduction

Operational transformation is an approach (Ellis & Gibbs 1989, Sun & Chen 2002) which allows to build real-time groupware like shared editors. Algorithms like aDOPTed (Ressel, Nitsche-Ruhland & Gunzenhauser 1996), GOTO (Sun, Jia, Zhang, Yang & Chen 1998), SOCT 2,3,4 (Suleiman, Cart & Ferrié 1998, Vidot, Cart, Ferrié & Suleiman 2000) are used to maintain the consistency of shared data. However these algorithms rely on the definition of transformation functions. If these functions are not correct then these algorithms cannot ensure the consistency of shared data.

Proving the correctness of transformation functions even on simple typed object like a String is a complex task. If we have more operations on more complex typed objects, the proof is nearly impossible without a computer. This is a serious

bottleneck for building more complex real-time groupware software.

We propose to assist development of transformation functions with the SPIKE theorem prover (Bouhoula & Rusinowitch 1995, Imine, Molli, Oster & Rusinowitch 2002). This approach requires specifying the transformation functions in first order logic. Then, SPIKE automatically determines the correctness of transformation functions. If correctness is violated, SPIKE returns counter-examples. As proofs are automatic, we can handle more (even complex) operations and develop quickly correct transformation functions.

This paper is organized in 3 sections. Section 2 briefly presents the transformational approach. In section 3 , we have verified with SPIKE the correctness of existing transformation functions with surprising results. Counter-examples provided by SPIKE helped us to define new *correct* transformation functions. Section 4 describes how to formalize transformation functions in SPIKE .

Transformational Approach

The model of transformational approach considers n sites. Each site has a copy of the shared objects. When an object is modified on one site, the operation is executed immediately and sent to others sites to be executed again. So every operation is processed in four steps:

(1) generation on one site,

(2) broadcast to others sites,

(3) reception by others sites,

(4) execution on other sites.

The execution context of a received operation op_i may be different from the generation context of op_i. In this case, the integration of op_i by others sites may leads to inconsistencies between replicates.

We illustrate this behavior in figure 1. There are two sites working on a shared data of type $String$. We consider that a $String$ object can be modified with the operation $Ins(p, c)$ for inserting a character c at position p in the String. We suppose the position of the first character in String is 1 (and not 0).

$user_1$ and $user_2$ generate 2 concurrent operations: $op_1 = Ins(2, f)$ and $op_2 = Ins(5, s)$. When op_1 is received and executed on site 2, it produces the expected String "effects". But, when op_2 is received on site 1, it does not take into account that op_1 has been executed before it. So, we obtain a divergence between sites 1 and 2.

In the operational transformation approach, received operations are transformed according to local concurrent operations and then executed. This transformation is done by calling transformation functions. A transformation function T takes two

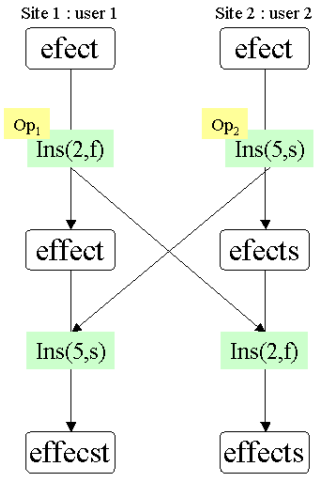

Figure 1. Incorrect integration

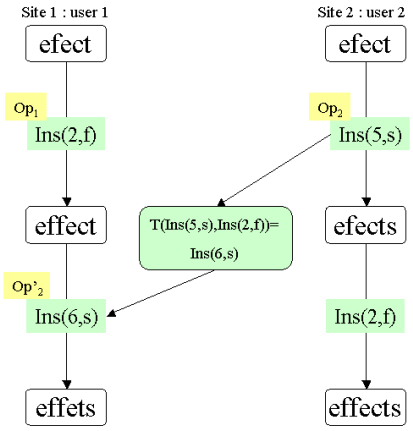

Figure 2. Integration with transformation

concurrent operations op_1 and op_2 defined on the same state s and returns op'_1. op'_1 is equivalent to op_1 but defined on a state where op_2 has been applied.

We illustrate the effect of a transformation function in figure 2. When op_2 is received on site 1, op_2 needs to be transformed according to op_1. The integration algorithm calls the transformation function as follows:

$$T(\overbrace{Ins(5,s)}^{op_2}, \overbrace{Ins(2,f)}^{op_1}) = \overbrace{Ins(6,s)}^{op'_2}$$

The insertion position of op_2 is incremented because op_1 has inserted an f before s in state $efect$. Next, op'_2 is executed on site 1. In the same way, when op_1 is received on site 2, the transformation algorithm calls:

$$T(\overbrace{Ins(2,f)}^{op_1}, \overbrace{Ins(5,s)}^{op_2}) = \overbrace{Ins(2,f)}^{op'_1}$$

In this case the transformation function returns $op'_1 = op_1$ because, f is inserted before s.

Intuitively we can write the transformation function as follows

---

T(Ins(p_1,c_1),Ins(p_2,c_2) :−
  if $p_1 < p_2$ **return** Ins(p_1, c_1)
  else return Ins($p_1 + 1$, c_1)

This example makes it clear that the transformational approach consists of two main components: the integration algorithm and the transformation functions. Integration algorithm are responsible of receiving, broadcasting and executing operations. It is independent of the type of shared data, it calls transformation functions when needed. The transformation functions are responsible for merging two concurrent operations defined on the same state. They are specific to the type of shared data (String in our example).

A lot of work has been done in defining a more theoretical model (Sun et al. 1998, Suleiman et al. 1998, Sun & Chen 2002, Sun 2002). Fundamentally, transformational approach defines a new consistency criteria for replicates. To be correct, an algorithm has to ensure three general properties:

Convergence When the system is idle (no operation in pipes), all copies are identical.

Causality If on one site, an operation op_2 has been executed after op_1, then op_2 must be executed after op_1 in all sites.

Intention preservation If an operation op_i has to be transformed into op'_i, then the effects of op'_i have to be equivalent to op_i.

To ensure these properties, it has been proved (Sun et al. 1998, Suleiman et al. 1998) that the underlying transformation functions must satisfy two conditions:

- The condition C_1 defines a *state equivalence*. The state generated by the execution op_1 followed by $T(op_2, op_1)$ must be the same that the state generated by op_2 followed by $T(op_1, op_2)$:

$$C_1 : op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

- The condition C_2 ensures that the transformation of an operation according to a sequence of concurrent operations does not depend of the order in which operations of the sequence are transformed:

$$C_2 : T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$$

It is important to note that although many algorithms have been defined, just few sets of transformation functions have been delivered to the community (Palmer & Cormack 1998, Davis, Sun & Lu 2002, Molli, Skaf-Molli, Oster & Jourdain 2002). Proving C_1 and C_2 on transformation functions is very long and error prone even on a simple string object. For example, there are 123 different cases to explore when trying to prove C_2 on a string object. Each time the specification of transformation functions is changed, it is necessary to redo the proof.

Without a correct set of transformation functions, algorithm cannot ensure consistency and resulting groupware tools will be error-prone. *To be able to develop the transformational approach with simple or more complex objects, proving conditions on transformation functions must be automatic.*

Verifying Transformation Functions

In this section, we return to existing transformation functions defined on String objects. We have formalized them using SPIKE and checked their correctness. We show in section 4 how to formalize these functions in SPIKE .

We consider a String to be an array of characters starting at range 1 (and not 0). Two operations are defined on String:

- $Ins(p, c)$: Inserts a character c at position p.
- $Del(p)$: Deletes the character located at position p.

Ellis Transformation Functions

Originally, Ellis and Gibbs (Ellis & Gibbs 1989) defined transformation functions as shown below. Operations $insert$ and $delete$ are extended with a new parameter pr representing the priority. Priorities are based on the site identifier where operations have been generated. $Id()$ is the Identity operation. It does not affect state.

T_{ii}(Ins(p_1,c_1,pr_1), Ins(p_2,c_2,pr_2)) :−
if $p_1 < p_2$ **return** Ins(p_1,c_1)

else if $p_1 > p_2$ **return** Ins($p_1 + 1$, c_1,pr_1)
else if $c_1 == c_2$ **return** Id()
else if $pr_1 > pr_2$ **return** Ins($p_1 + 1$,c_1,pr_1)
else return Ins(p_1,c_1,pr_1)

T_{id}(Ins(p_1,c_1,pr_1), Del(p_2,pr_2)) :−
if $p_1 < p_2$ **return** Ins(p_1,c_1,pr_1)
else return Ins($p_1 - 1$,c_1,pr_1)

T_{di}(Del(p_1,pr_1),Ins(p_2,c_2,pr_2)) :−
if $p_1 < p_2$ **return** Del(p_1,pr_1)
else return Del($p_1 + 1$,pr_1)

T_{dd}(Del(p_1,pr_1),Del(p_2,pr_2)) :−
if $p_1 < p_2$ **return** Del(p_1,pr_1)
else if $p_1 > p_2$ **return** Del($p_1 - 1$,pr_1)
else return Id()

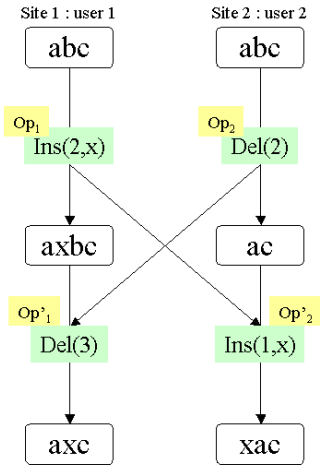

Figure 3. Counter-example violating condition C_1

It is well known that these transformation functions are not correct (Sun et al. 1998, Suleiman et al. 1998, Ressel et al. 1996). Nevertheless, they were submitted to SPIKE just to verify if the problem can be automatically detected. SPIKE found the counter-example depicted in figure 3 in just few seconds. SPIKE detected that condition C_1 is violated.

The counter-example is simple:

(1) $user_1$ inserts x in position 2 (op_1) while $user_2$ concurrently deletes the character at the same position (op_2).

(2) When op_2 is received by site 1, op_2 must transformed according to op_1. So

$T_{di}(Del(2), Ins(2,x))$ is called and $Del(3)$ is returned.

(3) In the same manner, op_1 is received on site 2 and must be transformed according to op_2. $T(Ins(2,x), Del(2))$ is called and return $Ins(1,x)$. Condition C_1 is violated, The final results on both sites are different.

The error comes from the definition of T_{id}. The condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$. But if we re-submit this version to the theorem prover, it is still not correct with the counter-example detailed in section 3.2 .

This is a typical example of working with SPIKE . In some way, we use it like a compiler. We express the type of functions using the SPIKE syntax and SPIKE checks conditions in few seconds or few minutes depending of the number of different cases induced by the specification.

Ressel Transformation Functions

Matthias Ressel et al (Ressel et al. 1996) have modified Ellis transformation functions in order to satisfy C_1 and C_2. Priorities are replaced by the parameter $u_i \in 1, 2, ..., n$. This parameter represents the user that generated the operation. Ressel wrote that T_{id} and T_{di} are exactly the same than Ellis. In this case, the set of transformation functions does not satisfy C_1 as in counter-example of figure 3. We suppose Ressel refers to a corrected version of Ellis where T_{id} is redefined with $p_1 \leq p_2$.

Compared to Ellis, Ressel modified the definition of T_{ii}. In case of insertion of 2 characters at the same position p, the character produced by the site with the lower range is inserted at p.

T_{ii}(Ins(p_1,c_1,u_1)),(Ins(p_2,c_2,u_2)) :−
if $p_1 < p_2$ or ($p_1 = p_2$ and $u_1 < u_2$) **return** Ins(p_1,c_1,u_1)
else return Ins(p_1+1,c_1,u_1)

T_{dd}(Del(p_1,u_1)),Del(p_2,u_2) :−
if $p_1 < p_2$ **return** Del(p_1,u_1)
else if $p_1 > p_2$ **return** Del($p_1 - 1$,u_1)
else return Id ()

T_{id}(Ins(p_1,c_1,u_1), Del(p_2,u_2)) :−
if $p_1 \leq p_2$ **return** Ins(p_1,c_1,u_1)
else return Ins($p_1 - 1$,c_1,u_1)

T_{di}(Del(p_1,u_1),Ins(p_2,c_2,u_2)) :−
if $p_1 < p_2$ **return** Del(p_1,u_1)
else return Del($p_1 + 1$,u_1)

This strategy seems to work but SPIKE found that the counter-example of figure 4 does not verify C_2.

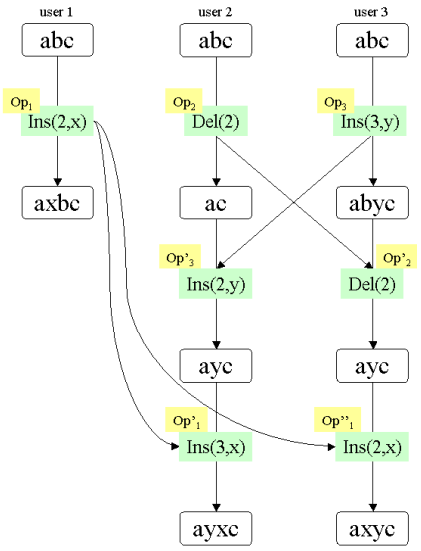

Figure 4. Counter example violating condition C_2

This counter-example requires 3 users. Operations $op_1 = Ins(2, x)$, $op_2 = Del(2)$, $op_3 = Ins(3, y)$ are concurrent.

(1) First of all, op_2 is integrated on $user_3$'s site. So we apply $T(Del(2), Ins(3, y))$ that return $op'_2 = Del(2)$.

(2) op_3 is integrated on site 2, we apply $T(Ins(3, y), Del(2))$ that return $op'_3 = Ins(2, y)$.

(3) Next, op_1 is integrated on site 2. op_1 must be transformed according to op_2 and the result of this transformation must be transformed according to op'_3. $T(op_1 = Ins(2, x), op_2 = Del(2))$ returns a new operation $Ins(2, x)$. This operation must be transformed again according to op'_3:

$$T(Ins(2, x), \overbrace{Ins(2, y)}^{op'_3}) = \overbrace{Ins(3, x)}^{op'_1}$$

(4) We do exactly the same for op_1 on site 3. So we calculate the result of:

$$\overbrace{Ins(2, x)}^{op''_1} = T(T(Ins(2, x), \overbrace{Ins(3, y)}^{op_3}), \overbrace{Del(2)}^{op'_2})$$

Copies on site 2 and 3 do not converge. Transformation functions of Ressel do not verify C_2.

Sun Transformation Functions

Chengzheng Sun (Sun et al. 1998) published the set of transformation functions below. The signature of operations $Insert$ and $Delete$ are little different. $Insert$ does not insert only a character at position p but a whole String s.

T(Ins(p_1,s_1,l_1),Ins(p_2,s_2,l_2)) :−
  if $p_1 < p_2$ **return** Ins(p_1, s_1, l_1)
  else return Ins($p_1 + l_2$, s_1, l_1)

T(Ins(p_1,s_1,l_1),Del(p_2,l_2)) :−
 if $p_1 \leq p_2$ **return** Ins(p_1,s_1,l_1)
 else if $p_1 > (p_2 + l_2)$ **return** Ins($p_1 - l_2$,s_1,l_1)
 else return Ins(p_2,s_1,l_1)

T(Del(p_1,l_1),Ins(p_2,s_2,l_2)) :−
 if $p_2 \geq p_1$ **return** Del(p_1,l_1)
 else if $p_1 \geq p_2$ **return** Del($p_1 + l_2$,l_1)
 else return [Del(p_1,$p_2 - p_1$);Del($p_2 + l_2$,$l_1 - (p_2 - p_1)$)]

T(Del(p_1,l_1),Del(p_2,l_2)) :−
 if $p_2 \geq p_1 + l_1$ **return** Del(p_1,l_1)
 else if $p_1 \geq p_2 + l_2$ **return** Del($p_1 - l_2$,l_1)
 else if $p_2 \leq p_1$ and $p_1 + l_1 \leq p_2 + l_2$ **return** Del(p_1,0)
 else if $p_2 \leq p_1$ and $p_1 + l_1 > p_2 + l_2$ **return** Del(p_2, $(p_1 + l_1) - (p_2 + l_2)$)
 else if $p_2 > p_1$ and $p_2 + l_2 \geq p_1 + l_1$ **return** Del(p_1, $p_2 - p_1$)
 else return Del(p_1, $l_1 - l_2$)

For a better comparison with others set of transformation functions, we have rewritten Sun functions with characters. The result is illustrated below.

T(Ins(p_1,c_1),Ins(p_2,c_2) :−
  if $p_1 < p_2$ **return** Ins(p_1, c_1)
  else return Ins($p_1 + 1$, c_1)

T(Ins(p_1,c_1),Del(p_2)) :−
 if $p_1 \leq p_2$ **return** Ins(p_1,c_1)
 else return Ins($p_1 - 1$,c_1)

T(Del(p_1),Ins(p_2,c_2)) :−
 if $p_1 < p_2$ **return** Del(p_1)
 else return Del($p_1 + 1$)

T(Del(p_1),Del(p_2)) :−
 if $p_1 < p_2$ **return** Del(p_1)
 else if $p_1 > p_2$ **return** Del($p_1 - 1$)
 else return Id ()

SPIKE found that this set of transformation functions violates C_2 with the counter-example presented in figure 5.

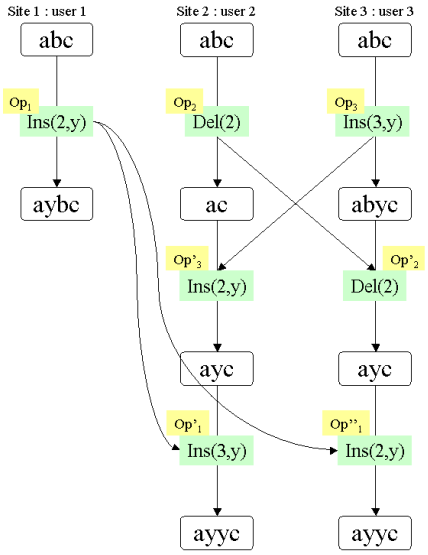

Figure 5. Counter example scenario that violates condition C_2

Consider 3 concurrent operations $op_1 = Ins(2, y)$, $op_2 = Del(2)$ and $op_3 = Ins(3, y)$.

(1) Site 3 integrates op_2.

$$\overbrace{Del(2)}^{op_2'} = T(\overbrace{Del(2)}^{op_2}, \overbrace{Ins(3, y)}^{op_3})$$

(2) Then, Site 2 integrates op_3.

$$\overbrace{Ins(2, y)}^{op_3'} = T(\overbrace{Ins(3, y)}^{op_3}, \overbrace{Del(2)}^{op_2})$$

(3) Next, Site 2 integrates op_1:

$$\overbrace{Ins(3, y)}^{op_1'} = T(T(\overbrace{Ins(2, y)}^{op_1}, \overbrace{Del(2)}^{op_2}), \overbrace{Ins(2, y)}^{op_3'})$$

(4) Finally, Site 3 integrates op_1:

$$\overbrace{Ins(2, y)}^{op_1''} = T(T(\overbrace{Ins(2, y)}^{op_1}, \overbrace{Ins(3, y)}^{op_3}), \overbrace{Del(2)}^{op_2'})$$

The final result is the same as in site 2 and 3, but C_2 is not satisfied. In fact:

$$\overbrace{Ins(3, y)}^{op_1'} = T(op_1, op_2 \circ T(op_3, op_2)) \neq \overbrace{Ins(2, y)}^{op_1''} = T(op_1, op_3 \circ T(op_2, op_3))$$

Note if we use $op_1 = Ins(2, x)$ instead of $op_1 = Ins(2, y)$, result will diverge on site 2 and 3 as in counter-example of figure 4.

Suleiman Transformation Functions

Suleiman (Suleiman, Cart & Ferrié 1997) proposes a different set of transformation functions. He adds two new parameters to function $Insert$: b_i is the set of operations that have deleted a character before the inserted character. a_i is the set of operations that have deleted a character after the inserted character.

So for two operations $Ins(p_1, c_1, b_1, a_1)$ and $Ins(p_2, c_2, b_2, a_2)$ defined on the same state:

- if $(b_1 \cap a_2) \neq \emptyset$ then c_2 was inserted before c_1
- if $(a_1 \cap b_2) \neq \emptyset$ then c_2 was inserted after c_1
- if $(b_1 \cap a_2) = (a_1 \cap b_2) = \emptyset$ then c_1 and c_2 were inserted at same position. So we can use the $code$ of character $code(c_i)$ to determine which character we have to insert at this position

---

T(Ins(p_1,c_1,b_1,a_1),Ins(p_2,c_2,b_2,a_2)) :−
  if $p_1 < p_2$ **return** Ins(p_1,c_1,b_1,a_1)
  else if $p_1 > p_2$ **return** Ins($p_1 + 1$,c_1,b_1,a_1)
  else // $p_1 == p_2$
    if $(b_1 \cap a_2) \neq \emptyset$ **return** Ins($p_1 + 1$,c_1,b_1,a_1)
    else if $(a_1 \cap b_2) \neq \emptyset$ **return** Ins(p_1,c_1,b_1,a_1)
    else if code(c_1)>code(c_2) **return** Ins(p_1,c_1,b_1,a_1)
    else if code(c_1)<code(c_2) **return** Ins($p_1 + 1$,c_1,b_1,a_1)
    else return Id ()

T(Ins(p_1,c_1,b_1,a_1),Del(p_2)) :−
  if $p_1 > p_2$ **return** Ins($p_1 - 1$,c_1,b_1+Del(p_2),a_1)
  else return Ins(p_1,c_1,b_1,a_1+Del(p_2))

T(Del(p_1),Del(p_2)) :−
  if $p_1 < p_2$ **return** Del(p_1)
  else if $p_1 > p_2$ **return** Del($p_1 - 1$)
  else return Id ()

T(Del(p_1,pr_1),Ins(x_2,p_2,b_2,a_2)) :−
  if $p_1 < p_2$ **return** Del(p_1)
  else return Del($p_1 + 1$)

SPIKE found that this set of transformation functions is correct. The only problem with it is the management of the sets a_i,b_i associated with each $Insert$ operations. The implementation is more difficult and transferring the $Insert$ operation is not efficient.

Imine Transformation Functions

We propose a new set of correct transformation functions simpler than Suleiman functions. In fact, Suleiman functions are over-specified. Managing the set of operations before and after each operation $Insert$ is not necessary. We propose to add a new parameter ip_i to every $Insert$ operation. This parameter represents the *initial position* of character c_i.

Suppose the user insert a character x at position 3, then an operation $Ins(3, 3, x)$ is generated. If this operation is transformed, only the position will change. The initial position parameter is not affected.

T(Ins(p_1,ip_1,c_1),Ins(p_2,ip_2,c_2) :−
  if p_1<p_2 **return** Ins(p_1,ip_1,c_1)
  else if p_1>p_2 Ins(p_1+1,ip_1,c_1)
  else // p_1==p_2
   if ip_1<ip_2 **return** Ins(p_1,ip_1,c_1)
   else if ip_1>ip_2 **return** Ins(p_1+1,ip_1,c_1)
   else // ip_1==ip_2
    if code(c_1) < code(c_2) **return** Ins(p_1,ip_1,c_1)
    else if code(c_1) > code(c_2) **return** Ins(p_1+1,ip_1,c_1)
    else // c_1==c_2
      return Id()

T(Ins(p_1,ip_1,c_1),Del(p_2)) :−
  if $p_1 > p_2$ **return** Ins($p_1 - 1$,ip_1,c_1)
  else return Ins(p_1,ip_1,c_1)

T(Del(p_1),Del(p_2)) :−
 if ($p_1 < p_2$) **return** Del(p_1)
 else if ($p_1 > p_2$) **return** Del($p_1 - 1$)
 else return Id()

T(Del(p_1,pr_1),Ins(p_2,ip_2,c_2)) :−
 if ($p_1 < p_2$) **return** Del(p_1)
 else return Del($p_1 + 1$)

After testing, SPIKE found that this set of transformation functions is correct. This kind of result shows an important issue of our approach. By studying counter-examples of Ellis, Ressel, we were sure that systems of priorities are unsafe. After proving that Suleiman functions were safe, we just tried to simplify them. With the theorem prover, it was easy for us to try different kind of simplification and finally converge on these transformation functions. This illustrate one important aspect of our approach. One serious bottleneck for developping transformation functions is the number of possible cases to be considered. With our approach, we delegate this

task to the theorem prover. So we can try a lot of different solutions in a short time. By this way, we have a process to develop quickly correct transformation functions.

Formalization of Transformation Functions

For modelling the structure and the manipulation of data in programs, Abstract Data Types (ADTs) are frequently used. Indeed, the *structure* of data is reflected by so called *constructors* (e.g., zero 0 and successor s may construct the ADT nat). Accordingly, all (potential) data are covered by the set of *constructors terms*, exclusively built by constructors. An ADT may have different *sorts*, each characterized by a separate set of constructors.

Furthermore, the *manipulation* of data is reflected by *function symbols* (e.g., $plus$ and $minus$ on nat). These symbols denote mappings over the elements of the data type. The intended properties of such mappings are specified by *axioms*, usually written in equational logic.

Model

More formally a real-time groupware system is a structure of the form $G =< S_t, O, Tr >$ where:

- S_t is the structure of the shared object (i.e., string, XML document, CAD object),
- O is the set of operations applied to the shared object,
- Tr is the transformation function.

In our approach, the shared object structure is transformed in ADT specification $State$. We define a sort Opn for the operation set O, where each operation serves as a constructor of this sort. For instance, a collaborative editing text has a character string as its shared object structure, and $O = \{O_1, O_2\}$ where:

- $O_1 = Ins(p, c)$ inserts character c at position p,
- $O_2 = Del(p)$ deletes the character at position p.

For the character string we may specify it with the ADT list; its constructors are $\langle\rangle$ and $l \circ x$ (i.e., an empty list and a list composed by an element x added to the back of list l respectively). Because all operations are mapped to the object structure in order to modify it, we give the following function:

$$\odot : State \times Opn \rightarrow State$$

All appropriate axioms of the function $\odot$ describe the transition between the object states when applying an operation. For example, the operation $Del(p)$ changes the character string as follows:

$$l \odot Del(p) \quad = \quad \begin{cases} \langle\rangle & \text{if} \quad l = \langle\rangle \\ l & \text{if} \quad l = l' \circ c \text{ and } p \geq |l| \\ l' & \text{if} \quad l = l' \circ c \text{ and } p = |l| - 1 \\ (l' \odot Del(p)) \circ x & \text{if} \quad l = l' \circ c \text{ and } p < |l| - 1 \end{cases}$$

where $|l|$ returns the length of the list l.

To overcome the user-intention violation problem, a transformation function is used in order to adjust the parameters of one operation according to the effects of other executed independent operations. Writing the specification of a transformation function in first-order logic is straightforward. For this we define the following function:

$$T : Opn \times Opn \rightarrow Opn$$

which takes two arguments, namely remote and local operations, and produces another operation. The axioms concerning this function show how the considered real-time groupware transforms its operations when they are broadcasted. As example, the following transformation:

T(Del(p_1),Ins(p_2,c_2) :−
if $p_1 > p_2$ **return** Del($p_1 + 1$)
else return Del(p_1)

---

is defined by two conditional equations:

$p_1 > p_2 \Longrightarrow$ T(Del(p_1),Ins(p_2,c_2)=Del($p_1 + 1$)
$p_1 \not> p_2 \Longrightarrow$ T(Del(p_1),Ins(p_2,c_2)=Del(p_1)

This example illustrates how easy it is to translate transformation function into the formalism of SPIKE . This task is straightforward and can be done automatically. The cost of formalisation is not expensive.

Specification of Conditions C1 and C2

We now express the convergence conditions as theorems to be proved in our algebraic setting. For this purpose, we use a predicate $Enabled : Opn \times State \rightarrow Bool$ expressing the condition under which an operation can be executed on a given state. Adding this predicate allows to avoid the generation of unreachable execution which violates conditions (Imine et al. 2002).

The first condition, C_1, expresses a *semantic equivalence* between two sequences where everything consists of two operations. Given two operations op_1 and op_2, the execution of the sequence of op_1 followed by $T(op_2, op_1)$ must produce the same tree as the execution of the sequence of op_2 followed by $T(op_1, op_2)$.

Theorem .1 (Condition C1).

$$\forall op_i, op_j \in Opn \textit{ and } \forall st \in State :$$

$$Enabled(op_i, st) \wedge Enabled(op_j, st) \Longrightarrow$$
$$st \circ op_i \circ T(op_j, op_i) = st \circ op_j \circ T(op_i, op_j)$$

The second condition, C_2, stipulates a *syntactic equivalence* between two sequences where everyone is composed of three operations. Given three operations op_1, op_2 and op_3, the transformation of op_3 with regard to the sequence formed by op_2 followed by $T(op_1, op_2)$ must gives the same operation as the transformation of op_3 with regard to the sequence formed by op_1 followed by $T(op_2, op_1)$.

Theorem .2 (Condition C2).

$$\forall op_i, op_j, op_k \in \text{Opn} :$$
$$Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Enabled(op_k, st) \Longrightarrow$$
$$T(op_k, op_i \circ T(op_j, op_i)) = T(op_k, op_j \circ T(op_i, op_j))$$

Conclusion and perspectives

We have illustrated in this paper the difficulty of having correct transformation functions. Just on a simple String object, all existing transformation functions are incorrect or over-specified. This problem come froms the difficulty to make proofs of correctness of transformation functions. On a simple String object, each time a function definition changes, you have to explore 100 different cases with caution. We are convinced that this task cannot be done correctly without the help of a computer. This approach is very valuable:

- The result is a set of safe transformation functions.
- During the development, the guidance of the theorem prover gives a high value feedback. Indeed, theorem prover quickly gives counter-examples.
- Formalization is easy.

We are convinced that this approach allows the transformational approach to be applied on more complex typed objects (Imine et al. 2002). We are working in several directions now:

- As we can prove C_1 and C_2 on large number of operations, we are currently developing correct transformation functions for a file system, XML files, blocks of text, etc. We are working not only on new sets of safe transformation functions but also on correctness of composition of these sets.
- We are currently modifying the SPIKE theorem prover in order to build an integrated development environment for transformation functions. Within this environment a user enters functions like in this paper and calls the theorem prover like a compiler. If there are errors, the environment gives counter-examples immediately. We believe that this kind of environment can greatly improve the process of production of transformation functions.

Acknowledgments

Many thanks for Fethi Rabhi, Senior lecturer at University of New South Wales of Sydney and Hala Molli, Associate Professor at University of Nancy, for their reviewing of this paper.

References

Bouhoula, A. & Rusinowitch, M. (1995), 'Implicit induction in conditional theories', *Journal of Automated Reasoning* **14**(2), 189–235.

Davis, A. H., Sun, C. & Lu, J. (2002), Generalizing operational transformation to the standard general markup language, *in* 'Proceedings of the 2002 ACM conference on Computer supported cooperative work', ACM Press, pp. 58–67.

Ellis, C. A. & Gibbs, S. J. (1989), Concurrency control in groupware systems, *in* 'SIGMOD Conference', Vol. 18, pp. 399–407.

Imine, A., Molli, P., Oster, G. & Rusinowitch, M. (2002), Development of transformation functions assisted by a theorem prover, *in* 'Fourth International Workshop on Collaborative Editing', New Orleans, Louisiana, USA.

Molli, P., Skaf-Molli, H., Oster, G. & Jourdain, S. (2002), Sams: Synchronous, asynchronous, multi-synchronous environments, *in* 'The Seventh International Conference on CSCW in Design', Rio de Janeiro, Brazil.

Palmer, C. R. & Cormack, G. V. (1998), Operation transforms for a distributed shared spreadsheet, *in* 'Proceedings of the 1998 ACM conference on Computer supported cooperative work', ACM Press, pp. 69–78.

Ressel, M., Nitsche-Ruhland, D. & Gunzenhauser, R. (1996), An integrating, transformation-oriented approach to concurrency control and undo in group editors, *in* 'Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)', Boston, Massachusetts, USA, pp. 288–297.

Suleiman, M., Cart, M. & Ferrié, J. (1997), Serialization of concurrent operations in a distributed collaborative environment, *in* 'Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge (GROUP'97)', ACM Press, pp. 435–445.

Suleiman, M., Cart, M. & Ferrié, J. (1998), Concurrent operations in a distributed and mobile collaborative environment, *in* 'Proceedings of the Fourteenth International Conference on Data Engineering (ICDE'98)', IEEE Computer Society, Orlando, Florida, USA, pp. 36–45.

Sun, C. (2002), 'Undo as concurrent inverse in group editors', *ACM Transactions on Computer-Human Interaction (TOCHI)* **9**(4), 309–361.

Sun, C. & Chen, D. (2002), 'Consistency maintenance in real-time collaborative graphics editing systems', *ACM Transactions on Computer-Human Interaction (TOCHI)* **9**(1), 1–41.

Sun, C., Jia, X., Zhang, Y., Yang, Y. & Chen, D. (1998), 'Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems', *ACM Transactions on Computer-Human Interaction (TOCHI)* **5**(1), 63–108.

Vidot, N., Cart, M., Ferrié, J. & Suleiman, M. (2000), Copies convergence in a distributed real-time collaborative environment, *in* 'Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'00)', Philadelphia, Pennsylvania, USA.